Shivam Rajendrakumar Dixit
CS 5800 Algorithms

Project Report

Title : **File Compression using Huffman Coding**
Syllabus Reference: Greedy Algorithms - **Huffman Coding Algorithm**
GitHub Link: https://github.com/dixitshiv/CS5800-Project/tree/main

1. Introduction

File compression is crucial in modern computing to optimize storage space, reduce transmission time, and enhance overall system performance. By eliminating redundancy in data representation, compression techniques significantly decrease the size of files while preserving their essential information. Huffman coding, developed by David A. Huffman in 1952, is a widely-used method for lossless data compression, particularly effective for text-based data such as documents, source code, and plaintext files. This report delves into the implementation of Huffman coding for file compression, shedding light on its underlying principles and implementation nuances

2. Huffman Coding Overview

Huffman coding operates on the principle of variable-length prefix coding, where more frequent characters are assigned shorter codes and less frequent characters are assigned longer codes. This ensures optimal compression by minimizing the average code length required to represent the data. The Huffman coding algorithm constructs a binary tree, known as the Huffman tree, where each leaf node represents a character and its frequency in the input data. The tree is built iteratively by merging the two nodes with the lowest frequencies until a single root node is formed. The resulting binary codes, obtained by traversing the Huffman tree from the root to each leaf, constitute the compressed representation of the input data.

3. Implementation Details

The provided code implements the Huffman coding algorithm using JavaScript. The HuffmanNode class represents a node in the Huffman tree, containing fields for the character, frequency, and references to its left and right children. The HuffmanTree class orchestrates the construction of the Huffman tree and provides methods for encoding and decoding data. The buildTree method constructs the Huffman tree based on the input character frequencies, while the encode and decode methods perform the compression and decompression operations, respectively. Additionally, the buildCodeMap method generates a mapping of characters to their corresponding Huffman codes, facilitating efficient encoding and decoding.

4. File Compression Process

The file compression process begins with analyzing the input data to determine the frequency of each character. Using this frequency information, the Huffman tree is constructed, capturing the hierarchical relationships among characters based on their frequencies. The input data is then encoded using the generated Huffman codes, replacing each character with its corresponding code. This results in a compressed representation of the original data, typically occupying fewer bits than the uncompressed data.

5. Decompression

Decompression involves reconstructing the Huffman tree from the compressed data, allowing for the reverse mapping of Huffman codes to their respective characters. The encoded data is traversed bit by bit, following the Huffman tree's structure to decode each code back to its original character. Once all codes have been decoded, the original data is recovered, identical to the input data before compression. Decompression is a vital aspect of Huffman coding, ensuring that the compressed data can be faithfully restored to its original form without any loss of information.

5. Code Explanation

```
class HuffmanNode {
  constructor(character, frequency) {
    this.character = character;
    this.frequency = frequency;
    this.left = null;
    this.right = null;
  }
}
```

The **HuffmanNode** class represents a node in the Huffman tree used for Huffman coding. Each node stores information about a character and its frequency in the input data.
character: Stores the character associated with the node.
frequency: Holds the frequency of occurrence of the character.
left and right: References to the left and right child nodes, respectively.
The constructor initializes a new HuffmanNode instance with provided character and frequency parameters, setting child references to null.

```
class HuffmanTree {
  constructor() {
    this.root = null;
  }

  buildTree(characters, frequencies) {
    const nodes = [];
    for (let i = 0; i < characters.length; i++) {
      nodes.push(new HuffmanNode(characters[i], frequencies[i]));
    }

    while (nodes.length > 1) {
      nodes.sort((a, b) => a.frequency - b.frequency);
      const left = nodes.shift();
      const right = nodes.shift();
      const parent = new HuffmanNode(null, left.frequency + right.frequency);
      parent.left = left;
      parent.right = right;
      nodes.push(parent);
    }

    this.root = nodes[0];
  }
```

The **HuffmanTree** class facilitates the construction of a Huffman tree, a crucial component of Huffman coding.

Constructor:
Initializes a new instance of the HuffmanTree class with an initial root node set to null.
buildTree(characters, frequencies):

- Constructs the Huffman tree based on the provided characters and their corresponding frequencies.
- Iterates through each character and its frequency to create individual leaf nodes.
- Sorts the leaf nodes based on their frequencies in ascending order.
- Iteratively merges the two lowest-frequency nodes into a single parent node until only one node remains, which becomes the root of the Huffman tree.
- The constructed Huffman tree is stored in the root attribute of the HuffmanTree instance.

```
encode(data) {
  const encodedData = [];
  const characterToCodeMap = this.buildCodeMap(this.root);
  for (let i = 0; i < data.length; i++) {
    encodedData.push(characterToCodeMap[data[i]]);
  }
  return encodedData.join("");
}
```

The **encode** method is responsible for encoding input data using the Huffman codes generated from the Huffman tree. Here's a breakdown of the method:

Input:
data: The input data to be encoded.
Process:
Initialization:

- Creates an empty array encodedData to store the encoded representation of the input data.
- Calls the buildCodeMap method to generate a mapping of characters to their corresponding Huffman codes.

Encoding:

- Iterates through each character in the input data.
- Retrieves the Huffman code corresponding to the current character from the characterToCodeMap.
- Appends the Huffman code to the encodedData array.

Return:
Returns the concatenated string representation of the encoded data using join("").
Which returns the encoded representation of the input data as a single string.

```
decode(encodedData) {
  const decodedData = [];
  let currentNode = this.root;
  for (let i = 0; i < encodedData.length; i++) {
    if (encodedData[i] === "0") {
      currentNode = currentNode.left;
    } else {
      currentNode = currentNode.right;
    }
    if (currentNode.character !== null) {
      decodedData.push(currentNode.character);
      currentNode = this.root;
    }
  }
  return decodedData.join("");
}
```

The **decode** method decodes the encoded data back to its original form using the Huffman tree. Here's a breakdown of the method:

Input:
encodedData: The encoded data to be decoded.
Process:
Initialization:

- Creates an empty array decodedData to store the decoded representation of the input data.
- Initializes a variable currentNode to the root of the Huffman tree.

Decoding:

- Iterates through each bit in the encoded data.
- Traverses the Huffman tree based on the current bit:
- If the bit is "0", moves to the left child node of currentNode.
- If the bit is "1", moves to the right child node of currentNode.
- Once a leaf node (character node) is reached, appends the corresponding character to the decodedData array and resets currentNode to the root.

Return:
Returns the concatenated string representation of the decoded data using join("").
Which returns the decoded representation of the encoded data as a single string.

```
  buildCodeMap(node, prefix = "") {
    const codeMap = {};
    if (node.character !== null) {
      codeMap[node.character] = prefix;
    } else {
      Object.assign(codeMap, this.buildCodeMap(node.left, prefix + "0"));
      Object.assign(codeMap, this.buildCodeMap(node.right, prefix + "1"));
    }
    return codeMap;
  }
}
```

The buildCodeMap method is responsible for generating a mapping of characters to their corresponding Huffman codes based on the Huffman tree. Here's a breakdown of the method:

Input:
node: The current node being processed in the Huffman tree.
prefix: The prefix code generated so far for the path from the root to the current node. It defaults to an empty string.
Process:
Initialization:
Initializes an empty object codeMap to store the mapping of characters to Huffman codes.
Base Case:

- Checks if the current node is a leaf node (i.e., it represents a character).
- If node.character is not null, it means the current node represents a character in the input data.
- Adds an entry to codeMap with the character as the key and the prefix as the value. This assigns the Huffman code for the character.

Recursive Case:

- If the current node is not a leaf node (i.e., it has children), recursively calls buildCodeMap for the left and right child nodes.
- Updates the prefix by appending "0" when traversing to the left child and "1" when traversing to the right child.
- Merges the resulting code mappings from the left and right subtrees into the codeMap using Object.assign.

Return:
Returns the codeMap a JavaScript object where each key represents a character, and its corresponding value is the Huffman code for that character

```
const data = "Algortihms is a great course";
const characters = Array.from(new Set(data.split("")));
const frequencies = characters.map(
  (char) => data.split("").filter((c) => c === char).length
);
const huffmanTree = new HuffmanTree();
huffmanTree.buildTree(characters, frequencies);
const encodedData = huffmanTree.encode(data);
console.log("Encoded data:", encodedData);

const decodedData = huffmanTree.decode(encodedData);
console.log("Decoded data:", decodedData);
```

The **data** variable contains the input text string to be encoded and decoded.

The **characters** variable is created by splitting the input data into individual characters and converting them into a set to remove duplicates. This ensures that each unique character is considered only once.

The **frequencies** variable is created by iterating over the unique characters and counting their occurrences in the input data. This generates an array of frequencies corresponding to each character.

Huffman Tree Construction:
An instance of the HuffmanTree class is created using const huffmanTree = new HuffmanTree();.

The **buildTree** method of the **huffmanTree** object is called with the characters and frequencies arrays as arguments. This constructs the Huffman tree based on the provided characters and their frequencies.

Encoding:
The encode method of the huffmanTree object is called with the input data as the argument. This generates the Huffman-encoded representation of the input data.

Decoding:
The decode method of the huffmanTree object is called with the encodedData as the argument. This decodes the Huffman-encoded data back to its original form.

Output:

```
PS C:\Users\HP> cd .\Desktop\
PS C:\Users\HP\Desktop> cd .\5800\
PS C:\Users\HP\Desktop\5800> node p3.js
Encoded data: 1110011101100010010011010101111110111110100111011010011110001110000011101110010100110000100100010010101101
Decoded data: Algortihms is a great course
PS C:\Users\HP\Desktop\5800>
```

6. Conclusion

In conclusion, Huffman coding offers an elegant and efficient solution for file compression, particularly well-suited for text-based data compression. This report has provided an in-depth exploration of Huffman coding, covering its fundamental principles, and implementation details. Through practical implementation and evaluation, I have demonstrated the viability of Huffman coding as a means of achieving lossless compression in real-world scenarios.

7. Thank You Note

To Professor,
I would like to express my sincere gratitude for your guidance and support throughout the duration of this course. Your expertise, encouragement, and invaluable feedback have been instrumental in my understanding of various concepts. Your passion for teaching and commitment to academic growth of all the students have truly made a difference in my learning experience.

To TA's,
I would also like to extend my heartfelt thanks to our teaching assistants for their dedication and assistance. Their patience, responsiveness, and willingness to assist with any challenges we encountered have been greatly appreciated. Their contributions have undoubtedly enriched our learning journey and helped us navigate this course more effectively.

As I conclude this project, I am grateful for the opportunity to apply theoretical concepts to practical implementation and experimentation. The knowledge and skills gained from this experience will undoubtedly benefit me in my academic and professional endeavors.

Thank you once again for your unwavering support and encouragement. I look forward to applying the insights gained from this course in my future studies and endeavors.

Warm regards,

Shivam Rajendrakumar Dixit